Videojuegos

Curso de Diseño y Programación

Nº 13 5,99 euros



Creación de nuestro tema musical

La cámara en Blitz3D

Realizar los elementos estáticos del decorado





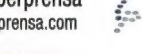




Curso práctico de Videojuegos

AUTOR DE LA OBRA
Marcos Medina
DIRECCIÓN EDITORIAL
Eduardo Toribio
etoribio@iberprensa.com
COORDINACIÓN EDITORIAL
Eva-Margarita García
eva@iberprensa.com
DISEÑO Y MAQUETACIÓN
Antonio Gª Tomé
PRODUCCIÓN
Marisa Cogorro
SUSCRIPCIONES
Tel: 91 628 02 03
Fax: 91 628 09 35
suscripciones@iberprensa.com

FILMACIÓN: Fotpreim Duvial
IMPRESIÓN: Gráficas Don Bosco
DUPLICACIÓN CD.ROM: M.P.O.
DISTRIBUCIÓN
S.G.E.L.
Avda. Valdelaparra 29 (Pol. Ind.)
28108 Alcobendas (Madrid)
Tel.: 91 657 69 00

EDITA: Iberprensa
www.iberprensa.com


CONSEJERO
Carlos Peropadre

REDACCIÓN, PUBLICIDAD Y ADMINISTRACIÓN
C/ del Río Ter, 7 (Pol. Ind. "El Nogal")
28110 Algete (Madrid)
Tel.: 91 628 02 03
Fax: 91 628 09 35
(Añada 34 si llama desde fuera de España.)

DEPÓSITO LEGAL: M-35934-2002
ISBN: **Coleccionable:** 84 932417 2 5
Tomo 2: 84 932417 4 1
Obra Completa: 84 932417 5 X

Aprende divirtiéndote

Bienvenidos de nuevo a **Programación y Diseño de Videojuegos**, la primera obra coleccionable cuyo objetivo es formar al alumno en las principales técnicas relacionadas en el desarrollo completo de un videojuego.

A lo largo de la obra el lector está aprendiendo programación a nivel general y a nivel específico con ciertas herramientas y lenguajes, aprendiendo a trabajar con aplicaciones de retoque de imagen y también de diseño 3D y animación. Estamos descubriendo las aplicaciones profesionales más importantes de audio y conociendo la historia de lo que se denomina "la industria del videojuego", los últimos 20 años, los juegos que marcaron un avance, sus creadores y en general la evolución del videojuego.

Pero además, esta obra tiene un segundo objetivo, desarrollar y potenciar la creatividad del lector, nosotros a lo largo de las diferentes entregas pondremos las bases y tú pondrás tu ingenio, tu creatividad y tu capacidad de mejorar.

Nos encontramos a mitad de camino del viaje de 20 semanas que os proponemos, viaje articulado en 400 páginas y 20 CD-ROMs cuya finalidad es proporcionar las bases mínimas para después cada uno continuar su camino.

Recuerda que para alcanzar el éxito necesitas cumplir tres condiciones: que te gusten los juegos, poseer cierta dosis de creatividad y finalmente capacidad de estudio.

Una la cumples seguro.

sumario



PARA ENCUADERNAR LA OBRA:

- *Para encuadernar los dos volúmenes que componen la obra "Programación y Diseño de Videojuegos" se pondrán a la venta las tapas 1 y 2.*
- ***Tapas del volumen 2 ya a la venta.***
- ***Los suscriptores recibirán las tapas en su domicilio sin cargo alguno como obsequio de Iberprensa.***

SERVICIO TÉCNICO:
Para consultas, dudas técnicas y reclamaciones Iberprensa ofrece la siguiente dirección de correo electrónico: games@iberprensa.com

PETICIÓN DE NÚMEROS ATRASADOS:
El envío de números sueltos o atrasados se realizará contra reembolso del precio de venta al público más el coste de los gastos de envío. Pueden ser solicitados en el teléfono de atención al cliente 91 628 02 03

Actuando con el entorno

Una vez completado el estudio de cómo controlar nuestra bionave de combate por el jugador, es preciso implementar un sistema de colisiones que proporcione una total interacción con el entorno.

Quizás ésta sea la parte más importante en el desarrollo de un juego, ya que puede determinar la credibilidad del mundo que queremos plasmar en el monitor. Así que vamos a intensificar nuestros esfuerzos en la detección, por parte de nuestra nave y de los disparos, del terreno, árboles, rocas y edificios, así como de los animales, plantas y otros enemigos.

(■) ¿CÓMO DESPLAZAR LA BIONAVE POR EL TERRENO?

No hace falta recalcar la importancia del terreno en un juego como "Zone of Fighters". En él se desarrolla toda la acción y es el soporte de todo el entorno. Es obvio que su presencia en el mapa de colisiones del juego es primordial.

Como explicamos en la sección de Blitz3D del número 8, lo que primero debemos hacer es crear un mapa de colisiones formado por las entidades involucradas. Para ello, es preciso asignar a cada una de ellas el tipo de colisión con el comando "EntityType". En realidad, se trata de establecer a cada entidad el número de identificación que tendrá dentro del mapa. Por ejemplo, podemos definir el terreno como entidad 1 o la bionave como entidad 4 y así con todas las demás:

```
EntityType terreno1, 1
EntityType camara, 2
EntityType bionave, 4
....
```

Realmente resulta sencillo. Pero supongamos que queremos definir un gran número de tipos de entidades. Llegará un momento en el que será difícil distinguir a qué tipo pertenece una entidad cualquiera cuando tengamos que hacer referencia a ellas en nuestro código. Para solucionar el problema, asignamos a cada número un nombre por medio de constantes. Así nuestra definición anterior quedaría:

```
Const ENTIDAD_TERRENO = 1
Const ENTIDAD_CAMARA = 2
Const ENTIDAD_BIONAVE = 4
EntityType terreno1, ENTIDAD_TERRENO
EntityType terreno1, ENTIDAD_CAMARA
EntityType bionave, ENTIDAD_BIONAVE
....
```

Como se puede comprobar, de esta manera es mucho más sencillo saber a qué tipo nos estamos refiriendo. Por lo tanto, en el módulo "Definiciones.bb" definiremos estos tipos de entidades mediante constantes:

```
Const ENTIDAD_TERRENO =  1
Const ENTIDAD_CAMARA  =  2
Const ENTIDAD_BIONAVE =  4
```

Una vez definidos los tipos debemos asignarlos para el mapa de colisiones (recordemos que la cámara es una entidad vinculada a la bionave, así que se comportará de la misma manera):

```
EntityType terreno1, ENTIDAD_TERRENO
EntityType terreno1, ENTIDAD_CAMARA
EntityType bionave, ENTIDAD_BIONAVE
```



La cámara se comporta exactamente igual que la bionave en el proceso de colisión con el terreno.

Para realizar todas estas definiciones creamos la función "Mapa_Colisiones", la cual situamos en el módulo de gestión del juego "Fjuego.bb" (Ver Fig.1).

Ya estamos preparados para activar la colisión entre ambas entidades mediante la instrucción "Collisions":

```
Collisions ENTIDAD_CAMARA,
ENTIDAD_TERRENO, 2, 3
Collisions ENTIDAD_BIONAVE,
ENTIDAD_TERRENO, 2, 3
```

Hemos utilizado como método el sistema de esfera a polígonos para que la bionave detecte el terreno. Es el más recomendable para utilizar con terrenos, ya que sólo tenemos que pasar al sistema de colisiones de Blitz3D el tamaño de la primera entidad. También hemos utilizado como respuesta al choque entre las dos entidades el sistema de deslizamiento. De esta forma, conseguimos que la bionave, en vez de quedarse parada al chocar con una elevación, se deslice suavemente. Ya sólo nos queda definir la esfera invisible que rodea a la bionave y que sirve de referencia al sistema de colisiones para poder realizar las detecciones con los polígonos del terreno.



NOTA

Todos los procesos de colisión que afecten a la bionave influirán de igual manera a la cámara como entidad ligada a ella.

Utilizando "EntityRadius" aumentamos el rango de acción del sistema de colisión "esfera - a - polígonos".

Utilizaremos el comando "EntityRadius" que aplicamos en el momento de cargar el modelo de la bionave en la función "crear_modelos" del módulo "funcpantaudio.bb" (Ver Fig.2):

```
bionave=LoadMesh ("c:\zone of fighters\modelos\bionave.3ds")
pivote_bionave=CreatePivot(bionave)
EntityTexture bionave,textura_bionave
EntityRadius bionave,55
...
```

Con todos estos procedimientos ya hemos logrado que la bionave no atraviese el terreno y que se deslice siguiendo sus irregularidades (Ver Fig.3).

(■) DETECTANDO EL DECORADO

Ahora mismo, si lanzamos a la bionave contra un edificio, un árbol o una roca, los atravesaría sin más. Esto es debido a que todavía no hemos definido su presencia en el mapa de colisiones del juego. Realmente, el paso para hacer esto es prácticamente igual que el realizado para la detección del terreno. En primer lugar definimos las constantes para los tipos de colisión del decorado. Así que podemos definir cada elemento anterior como sigue:

```
Const ENTIDAD_ALMACEN   = 5
Const ENTIDAD_ALMACEN2  = 6
Const ENTIDAD_PUENTE    = 7
Const ENTIDAD_GENERADOR = 8
Const ENTIDAD_ROCA1     = 9
Const ENTIDAD_ROCA2     = 10
Const ENTIDAD_ARBOL1    = 11
Const ENTIDAD_ARBOL2    = 12
...
```

Si observamos bien, estas definiciones no están verdaderamente optimizadas. Si nos paramos a pensar, veremos que lo que buscamos con la definición de colisiones es una respuesta determinada del sistema al choque de una entidad con otra. Según el diseño del juego sabemos que hay entidades del decorado que sufrirán algún tipo de desplazamiento cuando son alcanzadas por un disparo y otras que permanecerán intactas. Sabiendo este dato podemos deducir que en el juego existen dos tipos de entidades de decorado claramente diferenciadas, aunque para ambos tipos, la bionave se comportará de la misma forma en el momento de colisionar con ellos. Es evidente, también, que si la bionave colisiona con la entidad "Roca1" o "Almacen1" los efectos serán los mismos que si lo hiciera con la entidad "Roca2" o "Almacen2". Así que lo más lógico sería englobar a estas entidades en el mismo tipo de colisión denominándolas, por ejemplo, como del tipo "EDIFICIO". Así que podemos resumir la anterior definición de constantes de la siguiente manera:

```
Const ENTIDAD_EDIFICIO = 5
Const ENTIDAD_ARBOL    = 6
```

Todas las construcciones: almacenes, generadores, puentes y rocas, pertenecerán al grupo de los "edificios", el cual constituye el decorado estático. Y todos los tipos de árboles pertenecen al grupo de elementos del decorado móviles.

Por lo tanto, y siguiendo esta premisa, podemos reducir enormemente las definiciones y por consiguiente el mapa de colisiones para el resto de entidades. De esta manera obtenemos un mapa más limpio, pequeño y eficaz, lo cual favorece a obtener un mayor rendimiento en el juego. Ya sólo nos queda asignar estos tipos al mapa de colisiones y activarlos. Las asignaciones de los elementos estáticos y movibles del decorado la haremos en el momento de crear cada una de estas entidades. Así que trasladamos esta cuestión al próximo número, donde crearemos estos elementos. En la función "Mapa_Colisiones()" del módulo "Fjuego.bb" se encuentra el mapa completo de colisiones para "Zone of Fighters".

(■) ¿CÓMO AISLAR LAS COLISIONES?

Una vez determinado el mapa de colisiones de todo el juego debemos establecer las diferentes acciones derivadas del choque entre las entidades. Como sabemos, el mapa de colisiones nos sirve para activar todas las colisiones y obtener una respuesta determinada al choque, pero sólo eso. No sabemos, por ejemplo, con qué entidad en particular hemos colisionado ni podemos determinar una acción específica. Para ello, disponemos de una serie de instrucciones que utilizadas adecuadamente nos ayudarán en esta misión.



Para implementar un desplazamiento suave sobre el terreno es conveniente aplicar el método 3.



Se puede islar una entidad de un grupo del mismo tipo con "CountCollision" y "CollisionEntity".

Código 1. Detectamos con qué "volador" chocamos

```
If EntityCollided (bionave,ENTIDAD_ENEMIGO)
    If escudo_bionave=1 Return    ; Si el escudo está activado queda inmunizado, así que volvemos
    For n=1 To CountCollisions(bionave)
    ent=CollisionEntity(bionave,n) ; Guardamos con qué volador chocamos
    Next
    For volador.tipo_voladores= Each tipo_voladores
        If volador\entidad_volador=ent vida=vida-20 :play(schoque,c_schoque,bionave): Return
    Next
EndIf
```

En la función "actualizar_jugador_principal" del módulo "jugador_principal.bb" es donde resolvemos todas estas cuestiones. Antes de proseguir hay que recordar que los "voladores" son esos cubos que pululan por el escenario y que siempre persiguen a la bionave. En el mapa de colisiones los definimos como del tipo "enemigo", así que vamos a detectar con qué "volador" chocamos (Ver Fig.4) (Ver Código 1).

Una vez que hemos chocado con cualquier "volador" ("tipo ENEMIGO") utilizamos un bucle para recorrer cada una de las colisiones (almacenadas en la lista de colisiones) que ha sufrido la bionave durante la última actualización del mundo 3D con "CountCollisions". La otra entidad involucrada la guardamos en el manipulador "ent", el cual nos servirá posteriormente para averiguar el "volador" en concreto. Así que recorremos la estructura "tipo_voladores" y comparamos cada entidad de la estructura con la almacenada en "ent". Si coinciden, restamos vida a la bionave y volvemos. Hemos aprendido que lo importante es averiguar con qué entidad en concreto hemos colisionado, las consecuencias estarán determinadas por cada situación.

Para detectar los disparos enemigos no hace falta saber con cuál de ellos se ha colisionado ya que para todos se produce la misma respuesta.

Código 2. Aislamos cada tipo de armamento

```
If EntityCollided (disparo\sprite,ENTIDAD_TERRENO )
 Select tipo_armamento
  Case 3          ; Bombas de minifusión
   If Terreno_dinamico=True
    ex#=EntityX(disparo\sprite)
    ey#=EntityY(disparo\sprite)
    ez#=EntityZ(disparo\sprite)
    TFormPoint( ex,ey,ez,0,terreno1 )
    hi#=TerrainHeight( terreno1,TFormedX(),TFormedZ() )
    If hi>0
     hi=hi-.1:If hi<0 Then hi=0
     ModifyTerrain terreno1,TFormedX(),TFormedZ(),hi,True
     ModifyTerrain terreno1,TFormedX()+Rnd(-1,1),TFormedZ()+
     Rnd(-1,1),hi,True
     ModifyTerrain terreno1,TFormedX()+Rnd(-1,1),TFormedZ()+
     Rnd(-1,1),hi,True
    EndIf
   EndIf
   Crear_explosion(disparo)
   FreeEntity disparo\sprite
   Delete disparo
   Return
  Default
 End Select
Endif
```

(■) IMPLEMENTANDO LAS COLISIONES DE LOS DISPAROS

Quizás el proceso más interesante y complejo sea dar a conocer la presencia de los disparos en el juego al resto de entidades. Debemos implementar los procesos de colisiones en la función de actualización de cada disparo "actualizar_disparo" del módulo "jugador_principal.bb", ya que es la mejor forma de controlar las diferentes acciones por cada disparo individualmente.

(■) COLISIONANDO Y DEFORMANDO EL TERRENO

Según nuestro diseño inicial, se contempla la posibilidad de deformar el terreno para crear cráteres causados por el impacto de bombas de minifusión (Ver Fig.5).

De este modo, después de detectar la colisión debemos aislar cada tipo de armamento con una estructura "Select.. Case" (Ver Código 2).

ModifyTerrain terreno,TFormedX(),TFormedZ(),altura 5

Con el comando "ModifyTerrain" podemos obtener resultados como el que se muestra en la imagen.

Para poder modificar un terreno con "ModifyTerrain" es necesario averiguar las coordenadas exactas donde cayó el disparo por medio de "TFormPoint".

Para modificar el terreno utilizaremos la función "ModifyTerrain", la cual modifica la altura en un punto determinado de éste. Pero, ¿cómo podemos averiguar qué punto modificar? Esta información nos la proporcionarán las coordenadas del propio disparo. No obstante, para que todo surta efecto, es necesario trasladar estas coordenadas a las del terreno, las cuales son almacenadas por la función"TFormPoint".

A continuación, es preciso recuperar estos valores para que puedan ser utilizados por la función "ModifyTerrain". Para tal fin disponemos de las funciones "TFormedX", "TFormedY" y "TFormedZ" (Ver Fig. 6).

Después de formar el cráter, creamos una explosión, borramos la entidad disparo de la memoria y regresamos.

(■) COLISIONANDO CON EDIFICIOS Y ÁRBOLES

En las colisiones con los edificios (almacenes, puentes, generadores y rocas), además de crear la explosión, vamos a dejar una mancha. A este tipo de entidad la denominaremos "agujero" y es un sprite que orientaremos en la dirección del polígono que recibe el impacto.

Esta orientación la realizamos con la instrucción "AlignToVector", la cual alinea los ejes del sprite en la dirección del vector establecido por las normales del polígono impactado. Para poder orientar el sprite en cualquier dirección es necesario establecerle previamente el modo 4 de visualización con "SpriteViewMode" (Ver Código 3).

Código 3. Establecemos el modo 4 de visualización

```
If EntityCollided(disparo\sprite,ENTIDAD_EDIFICIO)
 If Decorado_interactivo=True
  For n=1 To CountCollisions(disparo\sprite )
   colision_x#=CollisionX(disparo\sprite,n )
   colision_y#=CollisionY(disparo\sprite,n )
   colision_z#=CollisionZ(disparo\sprite,n )
   normal_x#=CollisionNX(disparo\sprite,n )
   normal_y#=CollisionNY(disparo\sprite,n )
   normal_z#=CollisionNZ(disparo\sprite,n )
   ;Dejamos una mancha donde impacta el disparo para simular un agujero
   agujero.aguje=New aguje
   agujero\alfa=1
   agujero\sprite=CopyEntity(sprite_agujero)
   ScaleSprite agujero\sprite,Rnd(6,15),Rnd(6,15)
   PositionEntity agujero\sprite,colision_x,colision_y,colision_z
   AlignToVector agujero\sprite,-normal_x,-normal_y,-normal_z,3
   MoveEntity agujero\sprite,0,0,-.1
   Exit
  Next
 EndIf
 Crear_explosion(disparo)
 FreeEntity disparo\sprite
 Delete disparo
 Return
EndIf
```

Otro elemento del decorado que debemos detectar son los árboles. A diferencia de los edificios, no permanecerán estáticos cuando reciban el impacto de un disparo, sino que se moverán aleatoriamente unos pocos píxels en su eje X e Y:

```
If EntityCollided
(disparo\sprite,ENTIDAD_ARBOL)
 For n=1 To CountCollisions
 (disparo\sprite )
  ent=CollisionEntity
(disparo\sprite,n)
  xent#=EntityX(ent)
  yent#=EntityY(ent)
  zent#=EntityZ(ent)
  TurnEntity ent,Rnd(1,4),
  Rnd(1,4),0
 Next
...
```

Como podemos observar todos los procedimientos son prácticamente iguales entre sí.

Con este número hemos completado el estudio de la implementación de nuestra bionave de combate en el juego. El siguiente paso será dar vida al terreno de juego, controlando la manera de colocar todos los elementos del decorado según el sistema de juego seleccionado. Pero antes, sería interesante hacer un inciso para dedicar una entrega al sistema de partículas que nos permitirá crear efectos especiales como fuego o humo.



DEFINICIÓN

▸ VECTOR Y NORMAL

En un espacio 3D, un *vector* es un segmento de recta colocado en cualquier punto y con una dirección determinada. Una *normal* es un vector perpendicular a cualquier polígono.



En el próximo número...

... aplicaremos los efectos especiales de humo, fuego o climatología.

Fabricando los elementos del juego (IV)

Una vez que hemos terminado de crear los seres vivos del juego, pasamos a realizar los elementos estáticos del decorado (almacenes, generador, puente y rocas), así como los árboles pétreos.

MODELANDO LOS ALMACENES

En "Zone of Fighters" disponemos de dos tipos de almacenes, uno cúbico y otro pentagonal. Empezaremos modelando el primero de ellos con Milkshape3D.

Una vez preparadas las vistas pasamos a crear el cuerpo principal de la estructura, la cual basamos en un cubo. Para el techo, añadimos un par de cubos más, pero de menor altura y tamaño el primero y de igual altura pero de mayor tamaño el segundo. Este último cubo servirá para añadir una especie de visera, tal y como se muestra en la figura 1.

Continuamos, modelando y colocando las piezas de las esquinas. Para poder situarlas mejor, es conveniente ocultar el cubo mayor anterior (visera). Así que la seleccionamos en la pestaña *Groups* en el panel de herramientas y pulsamos en *Hide*.

Para las piezas de las esquinas partimos de un cubo escalado en forma de pilar. Para obtener la forma definitiva, seleccionamos los dos vértices exteriores de la parte superior y los unimos a los vértices interiores. Seguidamente, ajustamos la base de la pieza al cuerpo del almacén, desplazando los dos vértices interiores de la base hacia las paredes del cubo principal.

Para finalizar la pieza, unimos los vértices exteriores de la misma base entre sí ("Ctrl" + "N") (Ver Fig. 2).

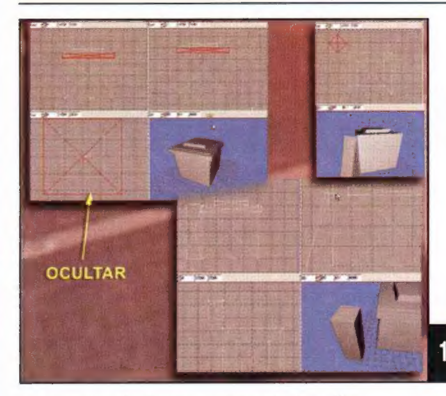
Para crear las otras piezas de las esquinas utilizamos funciones de duplicación y *Mirror*. Para poder continuar, volvemos a mostrar la pieza que ocultamos anteriormente. En ella, seleccionamos los vértices inferiores y los desplazamos para unirlos con la estructura principal del almacén. Terminamos, colocando la chapa en la pared donde irá posteriormente el logotipo del juego mediante un cubo escalado.

Para modelar el segundo almacén partiremos de un cilindro con 1 "Stack" y 7 "Slices" y con la opción *Close Cylinder* activada que servirá de cuerpo principal de la estructura. Para el techo, utilizamos duplicaciones escaladas de este cilindro tal y como se muestra en la figura 3.

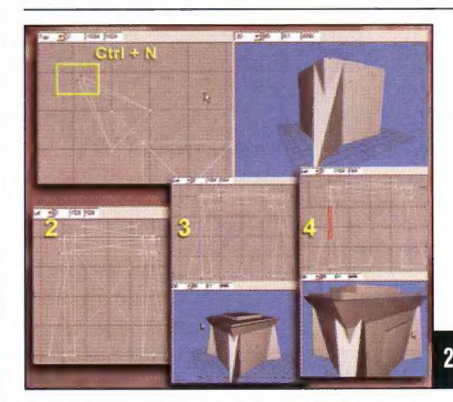
Para terminar el techo, seleccionamos los vértices de la cara superior de la última duplicación y los desplazamos hacia el interior con la herramienta de escalado. Finalizamos, colocando la pieza para situar el logotipo.

MODELANDO UN GENERADOR

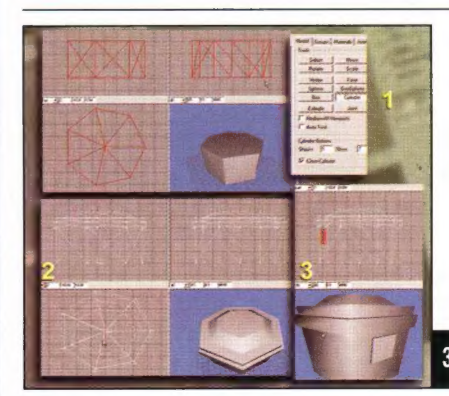
Los generadores son elementos más complejos de modelar. Constan de un cuerpo principal cilíndrico al que se le añade por ambos lados una gran tubería de tres piezas que llega hasta el suelo. Empezando con el cuerpo principal, situamos un cilindro de 1 "Stack" y 8 "Slices" y lo rotamos sobre su eje X 90 grados para volcarlo. Para



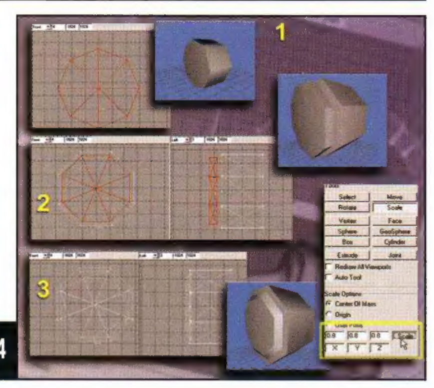
Para poder ajustar correctamente los pilares a cada esquina de la estructura principal es conveniente ocultar la pieza seleccionada de la figura.



Los vértices exteriores de la base del pilar se pueden unir para obtener la forma deseada.

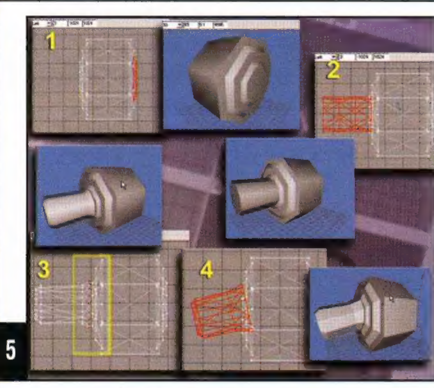


Duplicando y escalando el pentágono principal podemos obtener el resto de elementos del almacén.



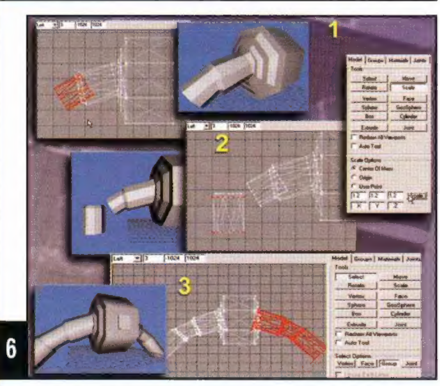
4

Duplicando el hexágono principal y escalando la copia obtenemos el anclaje mayor de las tuberías.



5

Las tres partes del cilindro que forman la primera sección de la tubería nos servirán para establecer la forma de los chaflanes de los extremos. Procedimiento 3 y 4 de la figura.



6

Debemos ajustar y rotar las demás secciones de la tubería para que alcance el suelo.

crear los anclajes de las tuberías utilizaremos el mismo cilindro, duplicándolo y aplicándole un escalado de 0.9 puntos en todos los ejes. De esta forma, nos ahorramos tener que rotar un cilindro nuevo. Seguidamente, seleccionamos los vértices exteriores de esta pieza y los escalamos hacia dentro en proporción 0.8. Realizamos las mismas operaciones para la siguiente pieza más pequeña del anclaje (Fig. 4).

Una vez terminado, seleccionamos las dos piezas y la duplicamos para crear el otro anclaje. Posteriormente, sólo tenemos que aplicar un *Mirror front <-> back* para situarlo al otro lado (Fig. 5 / 1).

El siguiente paso será crear una de las tuberías unidas al anclaje. Esta pieza consta de un cilindro situado en medio de otros dos mayores. Inicialmente, creamos el primer cuerpo de la tubería, el cual está unido a la estructura principal a través de uno de los anclajes. Para ello, partimos de un cilindro cerrado de 3 "Stacks" y 9 "Slices". Los "Stacks" o partes nos servirán para crear los ajustes de los extremos del trozo de tubería (Fig. 5 / 2).

Seleccionamos los vértices de cada parte y los desplazamos uno hacia el lado izquierdo y otro hacia el derecho. Posteriormente, seleccionamos los vértices de cada extremo y los escalamos manualmente hacia fuera (la parte interior) y hacia dentro (la exterior) o bien aplicando valores de 0.8 en cada eje. Obtendremos el resultado de la figura 5 / 3. Finalizaremos, girando un poco esta pieza para ir orientando la tubería hacia el suelo (Fig. 5 / 4). La siguiente parte está situada en el centro de la tubería y es de menor tamaño. Se realiza de la misma manera pero partiendo de un cilindro con sólo un "Stack", ya que no lleva uniones y queda incrustada en las otras dos piezas mayores. Seguidamente, volvemos a aplicar un pequeño giro para seguir la orientación hacia el suelo (Fig. 6 / 1).

Concluimos con la tercera parte de la tubería, que terminará introducida en la tierra. Por este motivo, sólo tendrá dos "Stacks", ya que sólo posee una unión en la parte interior (Fig. 6 / 2). Hasta el momento tenemos sólo una de las tuberías completa. Para realizar la del otro lado, como siempre, únicamente tenemos que seleccionarla por completo, duplicar y realizar un *Mirror front <-> back*. Y como es habitual, colocaremos un cubo escalado en el cuerpo principal para situar el logotipo del torneo (Fig. 6 / 3).

MODELANDO UN PUENTE

Esta pieza es un poco más compleja que la anterior y consta de dos partes diferenciadas: la base y techo del puente y los soportes del techo.

BASE DEL PUENTE

Esta parte de la estructura consta de tres piezas cúbicas. Situamos la primera que servirá de estructura principal donde estarán unidas todas las demás. Las siguientes dos piezas están colocadas justo bajo la principal y tienen la misma forma pero con menor tamaño. Para la segunda, haremos una duplicación y aplicamos un escalado con relación 0.9 en el eje X, 1.5 en el Y y 0.95 en el Z. La tercera pieza es una copia de la segunda pero con los vértices exteriores escalados hacia dentro, aplicando el valor 0.95 en cada eje (Fig. 7 / 1).

TECHO Y SOPORTES

Los soportes requieren un poco más de trabajo. Consiste en la consecución de pequeñas piezas cúbicas ligeramente inclinadas para formar un soporte curvo (Fig. 7 / 2 y Fig. 8 / 1).

Posteriormente, debemos unir cada uno de estos pequeños cubos entre sí. Para tal fin, uniremos los vértices superiores de la pieza de abajo con los vértices inferiores de la pieza de arriba, y así sucesivamente hasta unirlas todas (Fig. 8 / 2). El siguiente paso será realizar el acabado del soporte, estrechando los vértices superiores del extremo hasta conseguir el aspecto de la figura 9 / 1.

Los demás soportes se realizan por el procedimiento habitual de duplicado y aplicación de "mirrors" (Fig. 9 / 2).

Para terminar el puente sólo nos queda el techo, el cual es exactamente igual que la pieza principal del puente; es decir, un cubo que escalaremos hasta encajar con los cuatro soportes (Fig. 9 / 3).

MODELANDO LAS ROCAS

Estas estructuras adquieren tres formas diferentes, así evitamos feas repeticiones en elementos agrupados. Todas ellas se modelan de la misma manera, mediante la modificación de los vértices de un cilindro con más o menos partes y divisiones. Para la primera de las rocas partiremos de un cilindro cerrado con 2 "Stacks" y 8 "Slices" (Fig. 10 / 1).

A mayor número de partes y divisiones, mayor número de polígonos y complejidad de formas. Situándonos en la vista superior ("Top") seleccionamos todos los vértices de una división y los desplazamos hacia fuera. Hacemos lo mismo con la división adyacente (Fig. 10 / 2). De esta forma alargaremos la estructura de forma irregular. Para resumir, obtendremos muy diversas formas manipulando cada vértice o grupo de ellos en cada una de las divisiones. Podemos utilizar procedimientos de rotación, escalado o desplazamiento pero siempre siguiendo un orden y evitando la rotación excesiva de polígonos, ya que esto provocaría que sus caras no sean visibles (Fig. 10 / 3).

Para construir el resto de las rocas utilizaremos los mismos procedimientos. En estos casos es la imaginación en la manipulación de vértices la que origina las diferentes formas.

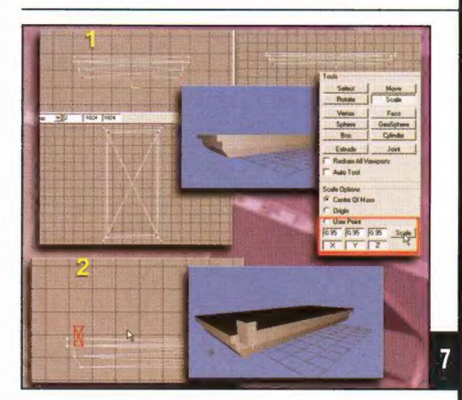
MODELANDO ÁRBOLES PETRIFICADOS

La creación de troncos para árboles es partícipe de la misma idea que utilizamos en el modelado de rocas. En este caso, podemos partir de un cilindro con 4 "Stacks" y 10 "Slices" (Fig. 11 / 1).

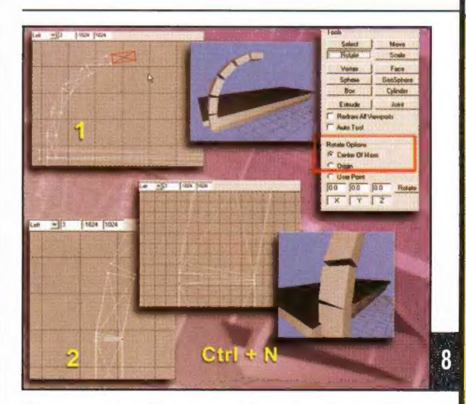
La idea radica en que el número de partes influye en la obtención de una forma de tronco más o menos compleja. Modificando los vértices de cada una de estas partes, por ejemplo, aplicando la herramienta de escalado, obtendremos formas variadas como troncos retorcidos. Lo primero que debemos modificar para crear un árbol son los vértices del primer "Stack" (parte), los cuales nos ayudarán a formar la base del árbol. El resto de vértices darán la forma del tronco (Fig. 11 / 2).

Para crear las ramas podemos realizar las mismas operaciones pero en cilindros más pequeños, los cuales uniremos al tronco posteriormente (Fig. 11 / 3).

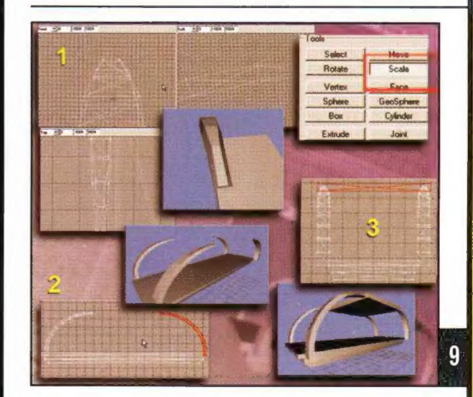
Después del modelado, el siguiente paso en la creación de un objeto 3D para un juego es el texturizado del mismo. Para las estructuras que hemos modelado en esta entrega, utilizaremos dos tipos de procedimientos de texturizado: pintando directamente en el modelo con Deep Paint 3D (generador) y por medio de plantillas ("templates") en un programa de dibujo como Paint Shop Pro (almacenes, puente, rocas y árboles). Dejaremos este último



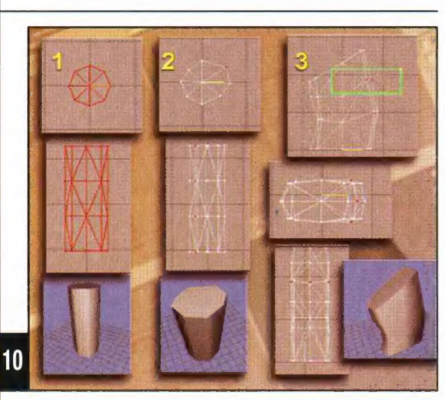
La base del puente consta de tres piezas iguales unidas, pero con diferente escalado.



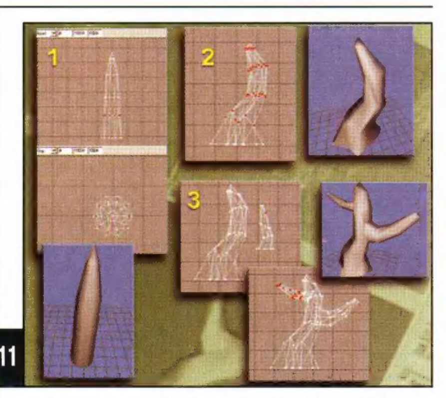
Una vez colocadas consecutivamente las secciones del soporte, se unen a través de los vértices. De esta manera es fácil obtener formas curvas con pocos polígonos.



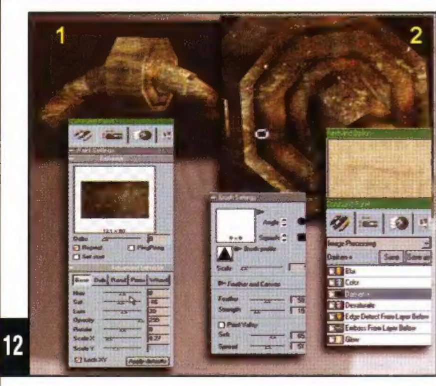
Mediante el escalado de vértices pares podemos estrechar estructuras.



A partir de un cilindro con más de tres partes se pueden obtener rocas de diferentes formas desplazando sus vértices.



La forma de modificar la posición de los vértices en cada sección de un cilindro determinará las diferentes estructuras.



Obtendremos más sensación de volumen en nuestro modelo si perfilamos con el pincel las uniones de sus partes con tonalidades oscuras.

procedimiento para el siguiente número del curso y nos centraremos ahora en el texturizado del generador.

CREANDO EL MAPEADO UV DEL GENERADOR CON LITHUNWRAP

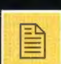
Como es habitual, antes del proceso de texturizado debemos crear un mapeado UV del modelo. Para el generador utilizaremos LithUnwrap. En primer lugar, cargamos el modelo salvado por Milkshape3D (.MS3D o .OBJ) (si no está a mano, se puede encontrar en "Extras" del CD llamado "generador.ms3d" o "generador.obj") Seguidamente, aplicamos un optimizado en *Tools/ Optimize models* para reducir polígonos. Seleccionamos todo con *Select all* y aplicamos un *UV Mapping Planar* con la opción *Planar X*. Terminamos el proceso, salvando de nuevo el modelo en formato .OBJ como "generador.obj".

TEXTURIZANDO EL GENERADOR CON DEEP PAINT 3D

Comenzamos cargando el modelo anterior. Al hacerlo aparecerá la ventana *Material Import*, indicando la presencia de un mapeado UV en el modelo y pidiendo la creación de un material nuevo para poder pintar sobre el modelo. Lo crearemos y seleccionaremos para él un tamaño de 256 x 256. Activamos el canal *C* con *nothing color* y ya estamos preparados.
Continuamos, asignando como vista de trabajo *Top* en *View*. Pulsamos dos veces seguidas sobre el icono de la herramienta de la lupa para ajustar el modelo a la ventana (para que surta efecto, debemos agrandar un poco manualmente el tamaño de la ventana de trabajo). Para rellenar el modelo seleccionaremos la textura *Metal Old 1 +* en la sección de texturas *Texture Paint*, a la que le modificaremos la saturación de color con un valor de –16 en *Sat*, la luminosidad a 30 en *Lum* y un escalado del eje X a 0.27 en *Scale X* (Fig. 12 / 1).

Una vez rellenado el modelo, vamos a utilizar el pincel para perfilar las uniones de las piezas del generador para distinguirlas. Aplicaremos un efecto de oscurecimiento de la imagen con pincel de tamaño 5. Este proceso lo seleccionamos de la sección *Imagen Processing* y escogemos *Darken +* (Fig. 12 / 2).

Una vez terminado el proceso de pintado, vamos a salvar la textura final. Así que nos dirigimos a la sección de materiales y pulsamos el botón derecho del ratón sobre la capa de color *C* y elegimos la opción *Export Channel*.



NOTA

**Un buen ejercicio para practicar el modelado con Milkshape3D sería realizar nuevos edificios, rocas o árboles o modificar los existentes.
Posteriormente, podríamos pintarlos con Deep Paint 3D y utilizar nuestro propio visualizador de objetos para ver los resultados, el cual desarrollamos en la sección de desarrollo del número 7 del curso. Incluso se podrían sustituir los ficheros del juego por los nuevos (siempre que se mantengan los mismos nombres) y de esta forma ver nuestros modelos en el editor de zonas de combate o en el propio juego.**

En el próximo número...

... realizaremos las texturas de los demás elementos del decorado por medio de plantillas con Paint Shop Pro.

Nuestro primer tema musical con Anvil **Studio** (II)

Continuando con la serie dedicada a Anvil Studio, comenzamos en este número el estudio de cómo realizar nuestro tema musical.

Como sabemos, en Anvil Studio podemos crear música mezclando pistas MIDI, de audio y de ritmos. Las pistas MIDI se pueden grabar utilizando un instrumento externo como un sintetizador, pero si no disponemos de ninguno, es posible hacerlo manualmente desde el *Compose* colocando nota a nota sobre el pentagrama. Para grabar pistas de audio, podemos utilizar un micrófono o a través de la entrada de línea de la tarjeta de sonido. Y para crear una pista de ritmo, sólo tenemos que elegir un sonido MIDI de la lista o insertar cualquier muestra.

CREANDO Y GRABANDO EN UNA PISTA MIDI

Vamos a crear una nueva canción, empezando por la grabación de una pista MIDI utilizando un instrumento externo conectado al ordenador a través de las conexiones MIDI de la tarjeta.

Por defecto, el programa se ejecuta con una pista de instrumento (MIDI) creada. Asi que sólo tenemos que seleccionarla para poder grabar en ella. Pero antes es necesario tener las conexiones correctamente. Para más detalles en la figura 1 se muestra una conexión típica de hasta dos teclados al ordenador.

A continuación, debemos asegurarnos de que el programa recibirá los eventos MIDI desde el teclado externo. Para ello, nos vamos al menú *View* y entramos en *Synthesizers*, en el cual se muestran los puertos de entrada y salida de audio y MIDI. En la casilla *MIDI In Port* debe aparecer *Maestro MPU-401* o *MIDI In*. Si queremos que suene el sintetizador externo debemos colocar en *MIDI Out Port* lo mismo que en el puerto de entrada. Si queremos que suene la tarjeta del ordenador elegimos los mapeadores MIDI o el sintetizador de la tarjeta de sonido. (Fig. 2 / 1).

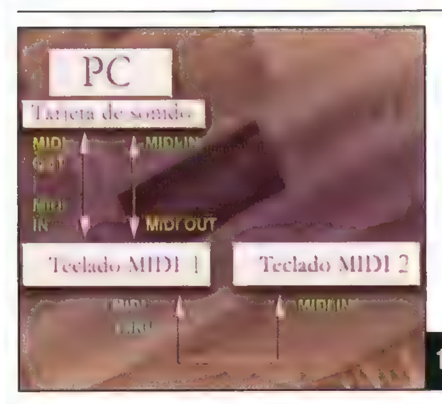
Es posible averiguar si todas las conexiones son correctas pulsando en el botón "Test MIDI Connections" (Fig 2 / 2).

Antes de grabar podemos también averiguar si todo marcha bien a través de la representación gráfica de luces que aparecen en el *Mixer* al lado del transportador. Estas "lucecitas" representan la actividad MIDI. Cada una de ellas interpreta un tipo de actividad cuando se torna de color rojo. En la figura 3 se muestra una tabla con el significado de cada una de estas actividades.

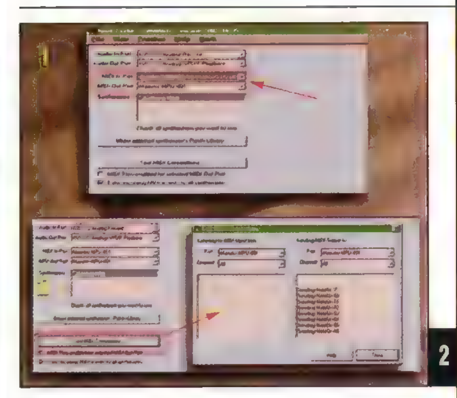
Bien, ya está todo preparado para grabar. Simplemente, pulsamos el botón de grabación "REC" y el programa hará una cuenta atrás antes de empezar a grabar. Sin embargo, antes de cualquier grabación es necesario preparar el tiempo y el compás, así como el metrónomo para acomodar la grabación a nuestras necesidades.

TIEMPO, COMPÁS Y METRÓNOMO

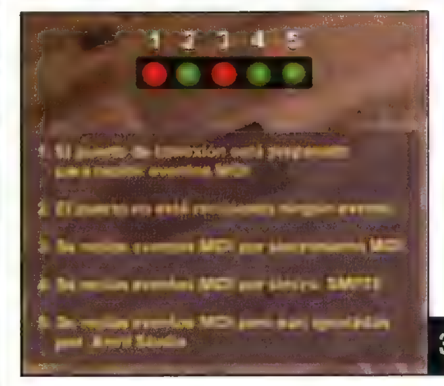
Para modificar los parámetros iniciales de nuestra canción, como la velocidad o el metrónomo, debemos entrar en la ventana de opciones



Esquema de una conexión MIDI de hasta dos teclados externos.



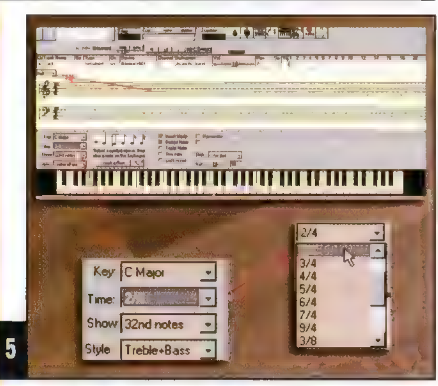
En Synthesizers del menú View podemos ajustar el puerto MIDI y realizar un chequeo de las conexiones.



Significado de la representación gráfica del estado de las conexiones MIDI.

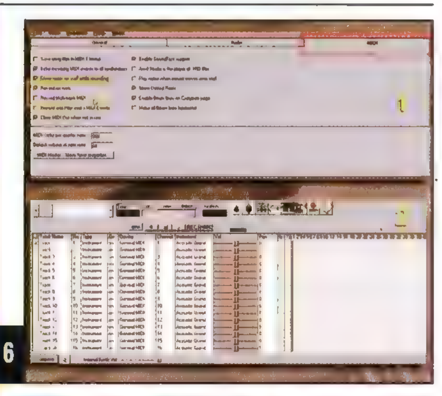
4

Descripción de la ventana de opciones del metrónomo y tiempo de la canción.



5

Procedimiento para cambiar el compás de una pista.



6

En las opciones MIDI del panel View / Options podemos activar la grabación MIDI simultánea en varias pistas.

Metronome / Tempo settings en el menú *View*. En primer lugar, encontramos la casilla *Tempo*, donde elegiremos el tiempo en pulsos por minutos. Seguidamente, las siguientes opciones se refieren al metrónomo. Resaltamos la posibilidad de cambiar su volumen o el sonido según un canal MIDI o nota determinada. En la figura 4 se muestra una descripción de esta ventana.

Para modificar el compás debemos hacerlo en la sección *Compose* en la forma de representación *Staff*. Una vez situados en esta sección podemos elegir el compás en la casilla *Time* (Fig. 5).

OPCIONES DE GRABACIÓN MIDI

Hay varias opciones muy interesantes que podemos activar para cambiar los aspectos de la grabación (Fig. 6 / 1).

Por ejemplo, se puede grabar en varias pistas MIDI a la vez. Para ello, nos situamos en la pestaña *MIDI* en la sección de opciones *View / Options*. Al volver al *Mixer* después de activarla observamos cómo automáticamente se han creado 16 pistas, una para cada canal MIDI, y todas con el mismo instrumento (Fig. 6 / 2). Al tocar, todos los eventos MIDI se grabarán por igual en todas las pistas.

Si estamos grabando desde *Compose* en el pentagrama *Staff*, es posible visualizar las notas al mismo tiempo que grabamos. Esta opción está situada también en las opciones *MIDI* y se denomina *Show notes on staff while recording*.

Una opción muy interesante se nos presenta también en las opciones MIDI denominada *Record No Rests*. Activándola, permitimos al programa que grabe exactamente como estamos tocando sin ningún tipo de cuantización en tiempo real. Si grabamos con esta opción activada es fácil que no ajustemos las notas a los compases perfectamente. En tal caso, siempre podemos aplicar una cuantización nosotros mismos. Para ello, seleccionamos la opción *Quantitize Entire Track* en el menú *Track*. Aparecerá una ventana flotante con la medida de cuantización.

OPCIONES ESPECIALES

Anvil Studio posee posibilidades muy interesantes a la hora de reproducir nuestras canciones. Por ejemplo, permite que el contenido de una pista suene con dos instrumentos a la vez. Evidentemente, para que esto sea posible, es necesario duplicar la pista y cambiar el canal MIDI. Una vez realizado, cambiamos el sonido de la pista duplicada. Para hacer más rápido este procedimiento, disponemos de una opción que divide la pista, creando una nueva idéntica. Generalmente se usa cuando tenemos una canción en una sola pista. Esta opción la podemos encontrar en el menú *Track* y se denomina *Split track into single channel track*. Como disponemos de hasta 16 canales MIDI, podemos tocar una misma pista con 16 instrumentos diferentes a la vez.

Para terminar, Anvil Studio admite posibilidades de transportación de notas para una sola pista o para la canción completa actuando en todas las pistas a la vez. Encontramos esta opción en el menú *Track / Transpose*. En ambas posibilidades el programa nos pedirá la cantidad de medios tonos que queremos transportar. De esta forma, con un valor de 12 subimos una octava completa y con uno de -12, la bajamos.

Hasta aquí nuestros primeros pasos para crear una canción.



En el próximo número...

... aprenderemos cómo editar en *Compose* entre otras cosas interesantes.

Manejo de cámaras con Blitz3D

Después de conocer, en el número anterior, la plasticidad que ofrece la animación de modelos en un mundo 3D, vamos a descubrir, con esta entrega, la definición de la cámara y sus posibilidades en Blitz3D.

Es importante entender el concepto de cámara en un mundo 3D. En 2D no es necesario este elemento, porque la imagen que vemos se extiende sólo en dos dimensiones; es decir, una superficie plana distribuida en los ejes X e Y. Por lo tanto, la cámara se mantiene fija en una misma posición sin posibilidad de cambio. Por el contrario, en 3D disponemos de un eje más, Z, con lo que es posible girar alrededor de la vista o cambiar la posición de la cámara. En 2D, la cámara se crea y coloca por defecto siempre en el mismo lugar; pero en 3D, necesitamos hacerlo nosotros mismos. En Blitz3D, la cámara se considera una entidad más y, además de mostrar el dibujado de la escena, nos permite controlar otros factores como el rango de visión, el "zoom" o efectos de niebla.



Con la función "CameraViewport" se puede modificar el tamaño horizontal y/o vertical de la vista de cámara.



NOTA

Es importante saber que la cámara sólo funciona en el búfer oculto (backbuffer).

DEFINICIÓN

▸ VIEWPORT

Una vista (viewport) es un área de la pantalla que corresponde a lo que la cámara ve .

CREANDO Y MANIPULANDO LA CÁMARA

Una vez establecido el modo gráfico, lo siguiente que debemos hacer para preparar nuestra escena 3D es crear al menos una cámara para visualizarla. Para realizar esta operación disponemos del comando "CreateCamera":

```
= CreateCamera
( [                ])
```

El parámetro opcional nos permite encadenar la cámara a una entidad cualquiera, por lo que si ésta se mueve, la cámara lo hará con ella.

Es evidente que el hecho de que Blitz3D trate a la cámara como una entidad más nos posibilita la aplicación de instrucciones de movimiento y rotación propia de entidades como "MoveEntity", "RotateEntity" o "TurnEntity". Así que podemos mover la cámara por un escenario, seguir a nuestro personaje en un juego o desplazar la cámara alrededor de un objeto por medio de la utilización de pivotes (Ver "ejemplo1.bb") (Fig. 1).

Otra opción muy interesante es que podemos crear cuantas cámaras queramos al mismo tiempo, con lo que obtendremos



La cámara es tratada como una entidad más, así que es posible desplazarla y rotarla con las funciones estándar como "MoveEntity" o "TurnEntity".



En la figura se muestra un efecto 16:9 estrechando la vista de cámara.



En Blitz3D podemos disponer de múltiples cámaras, cada una con una vista y efectos diferentes.



Mediante la función "CameraRange" podemos definir el rango de visualización de la cámara.

diferentes vistas en la pantalla de una misma escena.

CONTROL DE MÚLTIPLES VISTAS

Por defecto, toda la pantalla es considerada como la vista o "viewport" de la cámara; es decir, desde 0,0 hasta el tamaño horizontal y vertical que marque la resolución activa.

Sin embargo, el "viewport" puede ser modificado a voluntad para que ocupe un área determinada. Para ello, disponemos del comando "CameraViewport":

```
CameraViewport
```

Así, la pantalla completa estará definida de la siguiente manera:

```
CameraViewport
GraphicsWidth(),
GraphicsHeight()
```

De esta forma, si queremos tener nuestro mundo 3D visible exclusivamente en la mitad superior de la pantalla sólo tendremos que definir el "viewport" como sigue:

```
CameraViewport
,
GraphicsWidth(),
GraphicsHeight()/2
```

En el ejemplo 2 ("ejemplo2.bb") se puede observar cómo es posible cambiar el tamaño de la vista de la cámara en tiempo real (Fig. 2 y 3).

Blitz3D admite la posibilidad de crear varias cámaras a la vez; con ello se permite manipular más de una vista diferente. Un ejemplo de la aplicación de este sistema lo encontramos en programas 3D, en los cuales se muestran cuatro vistas diferentes de la escena (derecha, izquierda, superior e inferior). También se utiliza mucho en juegos de velocidad, donde disponemos de una vista para simular el espejo retrovisor del vehículo. Las utilidades son numerosas y Blitz3D no nos pone ninguna limitación al respecto, ya que podemos crear 2,10, 20 o más cámaras diferentes, cada una con su propia vista y efectos. La única barrera son las propias limitaciones del sistema. Evidentemente, la utilización de más de una cámara reduce el rendimiento general del programa. En el ejemplo 3 ("ejemplo3.bb") se muestra cómo se pueden crear y manipular a la vez numerosas cámaras cuyos manipuladores pertenecen a una estructura de datos (Fig. 4).

OPCIONES DE VISUALIZACIÓN

En ocasiones, necesitaremos controlar el rango de visualización de nuestra escena, por ejemplo, para limitar hasta cuánto podemos ver. Esto nos permitirá optimizar el rendimiento del "Z-Buffering". En Blitz3D podemos controlar este aspecto con la instrucción "CameraRange":

```
CameraRange ,
#,
#
```

El Z-BUFFER es una lista de números que almacena las propiedades de cada píxel de un polígono y su distancia a la cámara. Se utiliza en la técnica "Z-Buffering" para dibujar únicamente los polígonos que se ven. Otro utilidad del "Z-buffering" es la creación de efectos de niebla.

El parámetro"distancia_cercana" indica dónde comenzarán a dibujarse los objetos 3D enfrente de la cámara y "distancia_lejana" hasta dónde se dibujarán. Por defecto, se dispone de los valores 1, 1000; es decir: *CameraRange camara,1,1000*. Todos los objetos colocados antes y después de estos valores serán eliminados de la escena ("Clipping"). Controlando estos valores podemos aumentar o disminuir el rendimiento del programa, ya que determinamos la cantidad de polígonos que se dibujarán (Fig. 5, "ejemplo4.bb").

Sin embargo, el "Clipping" provoca un corte en la imagen que no es del todo atractivo a la vista. Para evitar el más cercano



DEFINICIÓN

▶ EL Z-BUFFER

El Z-BUFFER es una lista de números que almacena las propiedades de cada píxel de un polígono y su distancia a la cámara. Se utiliza en la técnica "Z-Buffering" para dibujar únicamente los polígonos que se ven. Otro utilidad del "Z-buffering" es la creación de efectos de niebla.



Situar la niebla antes del rango de visualización de la cámara ayudará a disimular el efecto "clipping".

se sitúa antes que el límite de dibujado para evitar, de este modo, visualizar el "Clipping". La instrucción "CameraFogColor" define un color para la niebla:

```
CameraFogColor
```

y "CameraFogMode" le indica al Blitz3D que la visualice:

```
CameraFogMode                 ,
True False
```



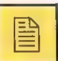
Podemos modificar el FOV de la cámara con la instrucción "CameraZoom".

se suele aplicar un valor en "distancia_cercana" lo más pequeño posible, como por ejemplo, 0.1, y para disimular el corte más lejano se aplica una efecto de niebla a una distancia inferior al valor "distancia_lejana". Para crear y utilizar un efecto de niebla determinado son necesarios varios pasos. En primer lugar, debemos definir el rango de visualización de la niebla; es decir, a qué distancia, frente a la cámara, estará situada. Seguidamente, le aplicaremos un color, y para terminar es necesario activar su visualización.

Para cada unos de estos procedimientos es primordial indicar a qué cámara se aplicarán, ya que, en caso de utilizar más de una a la vez, llevarán su propio "set" de efectos:

```
CameraRange
CameraFogRange
CameraFogColor
CameraFogMode           True
...
```

En el procedimiento anterior, se puede observar que la niebla



NOTA

No todas las tarjetas gráficas soportan efecto de niebla, y las que sí, ofrecen una calidad diferente.

(Fig. 6, "ejemplo5.bb").

Otras opciones muy interesantes para controlar el visualizado de la cámara consisten en aplicar un color de fondo de diferentes maneras y modificar el factor del "Zoom" (alejamiento y acercamiento). Si queremos borrar la vista con un color utilizaremos "CameraClsColor":

```
CameraClsColor
```

También podemos seleccionar el tipo de borrado que Blitz3D aplicará a la vista. Por un lado, podemos borrar el color y por otro los objetos 3D. Por defecto, los dos parámetros están activados:

```
CameraClsMode
```

En el ejemplo 4 se muestra el resultado de cambiar estos parámetros.

Para modificar el "Zoom" disponemos del comando "CameraZoom":

```
CameraZoom                    #
```

El parámetro "factor_zoom" es muy sensible y por defecto se sitúa en 1 (Ver "ejemplo5.bb"). Si aplicamos un valor entre 0 y 1 la cámara hará un "zoom out" (alejamiento) y si aplicamos valores superiores a la unidad un "zoom in" (acercamiento) El cambio del zoom de la cámara se conoce también como "FOV" (Fig. 7, "ejemplo6.bb").

FUNCIONES ESPECIALES

La comunicación entre los sistemas de coordenadas 2D y 3D es fundamental para una correcta implementación de la cámara en numerosas aplicaciones. Blitz3D nos proporciona todas las herramientas necesarias para este cometido. Así que podemos, por ejemplo, situar texto o elementos 2D en una escena 3D o permitir la interacción del ratón con objetos 3D, etc.

Imaginad que tenemos una escena 3D delante de la cámara compuesta de un terreno y tres objetos diferentes sobre él: un cubo, un cono y una esfera. Nuestro propósito es seleccionar cualquiera de estos objetos (incluido el terreno) mediante el puntero del ratón. Para ello, debemos utilizar dos funciones, una para los objetos que queremos seleccionar y otra para encontrarlos. Así que disponemos de la función "CameraPick", la cual nos dice con qué entidad (objeto) ha tocado el puntero del ratón:

```
CameraPick
```



Mediante la función "CameraPick" es posible la interacción entre el puntero del ratón y el entorno 3D.

3D, podemos obtener valiosa información sobre su posición y estado. Por medio de las funciones "PickedX#()", "PickedY#()" y "PickedZ#()" obtenemos las coordenadas X, Y y Z exactas donde tocó el puntero. Y con "PickedNX#()", "PickedNY#()" y "PickedNZ#()" obtenemos las componentes de las coordenadas X, Y,y Z de la normal.

Dependiendo de si es un mesh u otra entidad, la función "EntityInView" tomará distintas referencias del visualizado.

```
)
```

Pero para que esto funcione realmente, antes necesitamos decirle al Blitz3D qué objetos están dispuestos para ser detectados. Por lo tanto, debemos convertirlos en "pickeables"; es decir, que puedan ser tocados con el puntero del ratón. Así que es necesario aplicarles la función "EntityPickMode" con un valor de geometría que no sea 0 (Ver número 8 del coleccionable: "Blitz3D.Manejo de entidades"). Por ejemplo:

```
EntityPickMode
```

(Fig. 8, "ejemplo7.bb")
Una vez detectado el objeto

Además de las coordenadas, es posible averiguar hasta niveles más inferiores y precisos. Si tocamos con el puntero del ratón el cono de nuestro ejemplo, es posible, incluso, saber qué triángulo o superficie de la estructura del objeto hemos tocado. Esto es posible gracias a las funciones "PickedSurface()" y "PickedTriangle()". En ocasiones, podríamos necesitar saber cuánto tiempo se ha empleado en calcular cualquiera de estos comandos. Es muy útil, por ejemplo, para el testado y optimizado del código cuando estemos trabajando con este tipo de operaciones. Para averiguarlo, disponemos de la función "PickedTime()".

Volviendo a nuestra escena 3D anterior, queremos saber cuáles de los tres objetos (cubo, cono y esfera) quedan fuera de la vista de la cámara y cuáles no. Podemos averiguar esta información con la función "EntityInView". Esta función devolverá un 1 (True) si el objeto es visible y 0 (false) en caso contrario. Hay que tener en cuenta el punto de referencia que utiliza Blitz3D para realizar estas comprobaciones. Así, si nos referimos a un *mesh*, tomará como referencia la caja que lo limita y si es otra entidad, tomará como referencia su centro de posición (Fig. 9).

Ejemplo del funcionamiento de los métodos de proyección de coordenadas.

Por último, una interesante opción que encontramos entre las funciones de Blitz3D es la posibilidad de proyectar las coordenadas de un objeto 3D a coordenadas 2D. Esta opción es tremendamente útil, por ejemplo, en aplicaciones donde queremos imprimir un texto o número en la posición de un objeto 3D situado en cualquier lugar de la escena. Así que disponemos para ello de la función "CameraProject":

```
CameraProject
```

Esta función pasará los diferentes valores de coordenadas a las funciones "ProjectedX#()", "ProjectedY#()" y "ProjectedZ#()".

En el ejemplo 8 se puede observar el funcionamiento de estas funciones (Fig. 10).

La cámara es uno de los factores primordiales en cualquier escena 3D. Existe, sin embargo, otros cuyo estudio no deja de ser fascinante: la iluminación.



En el próximo número...

... estudiaremos el manejo de luces.

Utilización de imágenes para crear fuentes de letras

Como prometimos, vamos a desarrollar una serie de funciones que nos permitirán escribir letras o números en pantalla utilizando caracteres procedentes de imágenes predefinidas.

En primer lugar, debemos crear un alfabeto en una aplicación de dibujo como, por ejemplo, Paint Shop Pro. Nuestro alfabeto estará formado por letras de diseño; es decir, con los colores o formas que queramos. Seguidamente, aislaremos cada carácter y crearemos un fichero de imagen por cada uno de ellos en un formato cualquiera. Proponemos como mejor opción .PNG, ya que es un formato comprimido (como .JPG) y mantiene la calidad del .BMP. Además, posee canal alfa como los .TGA. Es un formato muy completo (Ver Fig. 1).

Cada uno de estos ficheros tendrá el nombre del carácter que corresponde; es decir, para la imagen de la letra "A" el fichero se llamará "A.png", y así sucesivamente.

Luego, en el programa, sustituiremos el carácter alfanumérico por su imagen correspondiente. De esta manera, podemos imprimir frases o números utilizando nuestro alfabeto de diseño sin estar limitados a las fuentes que tengamos instaladas en nuestro sistema.

Para aprovechar este procedimiento, crearemos una serie de funciones que utilizaremos para cada uno de los siguientes casos: imprimir una frase, imprimir una variable numérica o crear una entrada de datos o "Input". Todos ellos servirán perfectamente para implementar la impresión de nuestros caracteres especiales en cualquier situación de un juego: menú, diálogos, contadores (puntuación, tiempo, etc.) o entrada de datos (nombre del jugador, récord, etc.)

DEFINICIÓN DE DATOS Y FUNCIONES

Lo primero que debemos hacer es crear una matriz con los datos de cada letra del alfabeto de la "A" a la "Z" y de los números del "1" al "0" y otra equivalente para los gráficos correspondientes a cada letra. Pero antes, vamos a definir una constante para el número de caracteres de nuestro alfabeto:

```
Const Num_letras%=35 ; (36 ‡ 0 - 35)
```

Seguidamente creamos ambas matrices:

```
Dim Letra$    (Num_letras)
Dim Grafico (Num_letras)
```



Procedimientos para preparar un alfabeto de diseño.

Continuamos, rellenando la matriz de comparación con todas las letras de nuestro alfabeto a partir de un banco de datos. Aprovechamos también el bucle de lectura de datos para almacenar en la segunda matriz de gráficos las imágenes de cada carácter (Ver Código 1).

De esta forma, tenemos, por ejemplo, en "Letra$(0)" la letra "A" y en "Grafico(0)" la imagen de esta letra "A.png" y así sucesivamente. Después de cargar cada imagen la escalamos a un tamaño definido por las variables globales "SizeV#" y "SizeV#" con la instrucción "ScaleImage". De esta forma podemos controlar también el tamaño de las letras. Recordad que las variables globales anteriores tienen que ser definidas en el programa



TRUCO

La mejor manera de trabajar con las imágenes de nuestro juego es utilizar el formato .BMP. Una vez obtenidos los resultados finales, se pasarán a .PNG. Ahorraremos mucho espacio en disco y, llegado el momento, se tardará menos tiempo en su carga.

Código 1. Almacenamos los caracteres

```
Restore Datos_letras  ;Apuntamos a la base de datos
For n=0 To Num_letras
  Read Letra$(n)
  Grafico(n)=LoadImage(Letra$(n)+".png")
Next
; -----------------------------------------------------------
.Datos_letras
Data "A","B","C","D","E","F","G","H","I","J","K","L","M"
Data "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"
Data "1","2","3","4","5","6","7","8","9","0"
```



2

Ejemplo de impresión de una frase utilizando imágenes en vez de fuentes de letras.

principal que llama a estas funciones.

Una vez definidas las matrices podemos proceder a crear las diferentes funciones.

FUNCIÓN PARA IMPRIMIR UNA FRASE

Esta función toma como dato de entrada la frase que queremos imprimir y la posición en la pantalla donde lo hará:

```
Function Imprimir_frase
( Frase$, x#, y#).
```

Ya dentro de la función, convertimos todos los caracteres de la frase entrante en mayúsculas antes de entrar en el bucle de búsqueda, conversión e impresión:

```
Frase$ = Upper ( Frase$ ).
```

Este bucle recorrerá cada uno de los caracteres de la frase y los comparará con los de la matriz de letras. Si coinciden, se imprimirá en pantalla la imagen del carácter correspondiente:


3

Función para imprimir una variable numérica utilizando un alfabeto de diseño.

```
For n = 1 To Len ( Frase$ )
 For m=0 To Num_letras
  If Mid(Frase$,n,1)=Letra$(m)
   GLetra=Grafico(m)
   DrawImage GLetra,x,y
   x=x+ImageWidth(GLetra)
   If x>ResolucionX-ImageWidth(GLetra)
    y=y+ImageHeight(GLetra)
    x=0
   EndIf
  EndIf
 Next
Next
```

Dentro del bucle principal que recorre toda la frase entramos en otro que recorrerá nuestra matriz de letras. Con la instrucción "Mid" aislamos el carácter correspondiente de la frase entrante y lo comparamos con cada letra de nuestra matriz. Cuando coincidan, imprimimos en pantalla su imagen correspondiente controlando, claro está, su posición en la pantalla gráfica (Ver "ejemplo1.bb") .

FUNCIÓN PARA IMPRIMIR UNA VARIABLE NUMÉRICA

Esta función servirá para imprimir cualquier variable numérica (en nuestro caso sólo números enteros) como puede ser un contador de puntuación o de tiempo con nuestro alfabeto de diseño.

Básicamente funciona igual que la función anterior a diferencia que el dato de entrada "Frase$" se sustituye por un valor numérico "Num%". Para poder imprimir los caracteres numéricos correspondientes, es preciso convertir el número entrante en alfanumérico con la función "Str":

```
Function Imprimir_Numero
( Num%, x, y )
   Numero$ = Str Num
```

El resto de la función es exactamente igual (Ver "ejemplo2bb").



FUNCIÓN DE ENTRADA DE DATOS

En esta función se va almacenando en una variable global alfanumérica cada carácter introducido desde el teclado con la instrucción "GetKey()". A medida que los caracteres se van introduciendo, se imprimen en pantalla. Saldremos de la función al pulsar la tecla de retorno "Return". En el "ejemplo3.bb" se muestra el modo de utilizar esta función. La variable global que almacenará la frase completa es "Entrada$".



4

Función para la obtención de datos desde el teclado utilizando el sistema de impresión de caracteres gráficos.

En el "ejemplo4.bb" se muestra este sistema de impresión de caracteres gráficos en un entorno 3D, el cual podría ser cualquier videojuego.

Este tipo de técnica resulta muy útil para dar mayor riqueza gráfica a nuestros juegos. Utilizada adecuadamente, es fácil modificar todos los textos de nuestro juego a diferentes idiomas.


5

Resultado de un ejemplo práctico del sistema de impresión de caracteres gráficos para juegos.

En el próximo número...

... veremos una técnica eficaz para crear y organizar una tabla de récords.

Juegos de estrategia (yIV). juegos divinos y de **puzzles**

Terminamos la serie dedicada a los juegos de estrategia en tiempo real, hablando de un subgénero muy especial con mezclas de simulación y que ha suministrado títulos de gran espectacularidad, belleza y que, sobre todo, ha proporcionado una forma de jugabilidad inigualable.

Nos referimos a los juegos divinos. Aunque en el número anterior vimos juegos como la serie *RailRoad* o la serie *Theme Park*, que pueden estar incluidos en este subgénero, hemos querido hacer un estudio aparte del trabajo de Peter Molyneux, el padre de los "God-Games". También hablaremos de un género de estrategia que nos ha acompañado desde los primeros tiempos de la historia del videojuego: los juegos de puzzles.

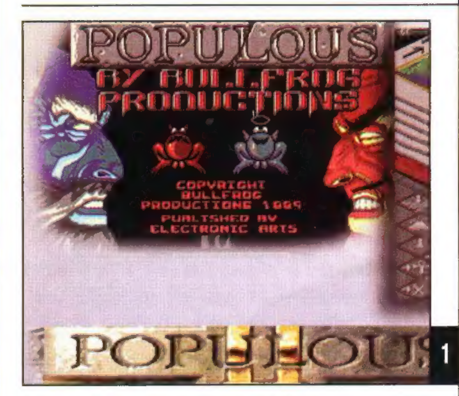
SIMULADORES DE DIOS

Encontramos un género, mezcla de simulación y estrategia, que levantó pasiones desde la publicación por Electronic Arts de *Populous* en 1989; un juego desarrollado por Bullfrog de la mano del gurú del entretenimiento Peter Molyneux. El concepto que mostraba *Populous* era convertir al jugador en una deidad. El cometido principal era hacer más confortable la vida del pueblo para así obtener poderes y destruir a los demás dioses, quienes tenían los mismos objetivos. Como dios, el jugador poseía algunos poderes para destruir como fuego, terremotos, huracanes, etc. Ese mismo año aparecía *Populous: The Promised Lands* con más de lo mismo. La saga se extendió y a lo largo de la década se desarrollaron nuevos títulos como: *Populous II: Trials of the Olympian Gods* (1993) o *Populous 3: The Beginning* (1998), el cual, y siguiendo las líneas tecnológicas del momento, pasa de la isometría a un entorno completamente 3D y donde el jugador se convierte en el dios de un pequeño planeta en el que no sólo controla a sus habitantes sino la Tierra. Un año después, aparece una expansión de *The Beginning* llamada *Undiscovered Worlds* (1999), en la que la historia se basa en la llegada al cielo para restablecer el equilibrio en el mundo de los dioses.

Antes de la publicación de *Populous II*, la casa de Molyneux desarrolla *Dungeon Keeper* (1997), donde el jugador tiene que construir y controlar un mundo bajo tierra aterrorizando a sus habitantes y destruyendo enemigos. La segunda parte se publicó dos años después con mejores gráficos pero con el mismo argumento.

Molyneux deja Bullfrog en 1997 y funda LionHead Studios para crear el God-Game más espectacular y original jamás visto hasta entonces: *Black & White (2001)*

El concepto impuesto para *Black & White* supuso un revulsivo en la industria del videojuego. Su argumento inicial seguía siendo el papel de dios que asume el jugador. En el juego existe una combinación única entre estrategia, construcción de mundos y simulación social con una línea argumental basada en juegos de rol que permite al jugador ser un dios bueno o malo encarnado en una gran criatura. Nuestra criatura es capaz de aprender y de seguir patrones independientes de comportamiento. Posee un gran nivel técnico con aspectos gráficos como: zoom infinito, física y climatología real o luces y sombras dinámicas. Además, el juego está conectado con la vida real; así, por ejemplo, podemos asignar a una criatura para que lea nuestro correo electrónico. Pero sobre todo es su IA



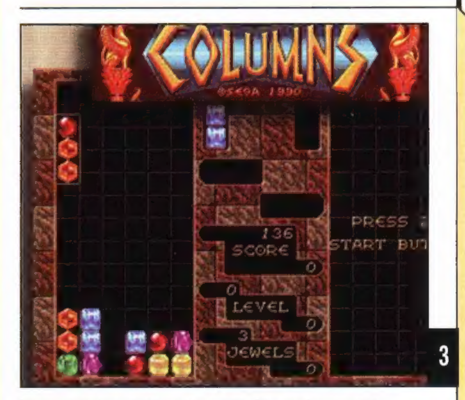
1

La serie Populous, pionera del género God-Games.



2

Black & White revolucionó la forma de diseñar juegos de estrategia.



3

Tras el éxito de Populous, Bullfrog desarrolló Dungeon Keeper, donde el concepto de juego divino se traslada a un mundo subterráneo.

LA BIOGRAFÍA...

PETER MOLYNEUX

Creador del **God-Game**

Considerado como uno de los mejores creadores de juegos de todos los tiempos, Peter Molyneux empezó programando hojas de cálculos. Entró en el mundo de los juegos para ordenador con *Fusion* en 1987, un juego de acción que provocó la creación de la empresa Bullfrog. Con Bullfrog desarrolló un nuevo concepto de jugabilidad que plasmó en *Populous* y en el que ponía al jugador en el papel de un dios. A raíz de este título Molyneux se consagró como unos de los más originales diseñadores de juegos y pronto su empresa creció sin límites. Siguió desarrollando God-Games como *Dungeon Keeper, Syndicate* o *Theme Park*, sin olvidar la serie *Populous*. En 1997 deja la compañía Bullfrog después de ser comprada por Electronic Arts y funda LionHead Studios en 1997. Su nuevo equipo se queda perplejo ante la nueva idea que Molyneux tenía entre manos: *Black & White*. Con este título, alcanza la definición absoluta de "juego divino". Y su nueva empresa se convierte en el núcleo de otras compañías afiliadas como: Big Blue Box Studios, Black & White Studios e Intrepid Games, los cuales desarrollan sin parar las nuevas invenciones de este gran Gurú del sector.



Peter Molyneux

lo que sobresale del juego. Los habitantes de *Eden* tienen vida propia y son capaces de simular alegría, tristeza, gratitud o temor además de la posibilidad emprendedora típica del ser humano. No podemos olvidar la interfaz de usuario única en el que el jugador sólo tiene que manejar el puntero del ratón convertido en mano.

Posteriormente a este título aparecieron nuevas partes como *B&W: Criature Isle, B&W: Next Generation* y en curso *Black and White 2*.

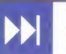
JUEGOS DE PUZZLES

Terminamos este capítulo dedicado a los juegos de estrategia con el género más conocido y popular de todos. Los también denominados "estruja cerebros" o juegos de habilidad mental. Prácticamente, los juegos de puzzles han sido el tipo de estrategia que ha dominado todos los tipos de consolas, ya que precisan de pocos recursos gráficos como aceleración 3D o sonido envolvente. Este tipo de juegos era el favorito en las recreativas de los años 80. La sencillez de gráficos necesarios para lograr un título más o menos aceptable produjo una avalancha de producción para todas las consolas del momento ya que prácticamente, los títulos de consolas se apoyaban en juegos de puzzles para reunir a la familia en torno al televisor. La mayoría de estos juegos para consolas han sido versionados para PC o están disponibles a través de emuladores especiales.

En realidad, se podría hablar de un antes y un después del clásico *Tetris* de Alexey Pajitnov (1985). Todos los juegos posteriores a este título están, de alguna manera, influenciados en su sólido concepto. La idea básica de este género es mostrar al jugador numerosos rompecabezas basados en reorganizar una serie de formas y colores en un patrón predefinido. Antes de la salida al mercado del *Tetris*, entre 1976 y 1987 existieron títulos de tipo educacional, como *Othello* o *Tic-Tac-Toe* (1977), basados en juegos populares. Pero los que más sobresalieron en la época fueron: *Maze* (1976), *Qix* (1981), *QBert* (1982) o *Locomotion* (1982).

Al publicarse *Tetris* para ordenadores personales en 1987, todo cambió. El concepto de videojuego de puzzle adquirió la premisa básica de encajar piezas de diferentes colores y formas entre sí. Está basado en la utilización de 7 figuras formadas por 4 pequeños bloques unidos de diferente forma (la palabra "tetris" viene de "tetra", 4). *Tetris* ha sido el juego más versionado de todos y el único que ha aparecido en todas las formas de máquinas de juegos conocidas, desde las recreativas hasta la consola de llavero. Después del *Tetris*, todas las desarrolladoras de equipos lúdicos tenían su propia versión. Algunas de ellas optaron por representar el "mundo tetris" en 3D (*Block Out*, 1989) o simulado (*Klax*, 1989) Pero pocos llegaron desarrollar un estilo personal y original como en el caso de *Ishido* (1990), *Bust-A-Move* (1994), *Panic Bomber* (1994) o *Columns* (1990), este último con una jugabilidad incluso superior, ya que permitía muchas más combinaciones en el juego. También aparecieron nuevos conceptos utilizando otras maneras de usar las formas y colores mediante imágenes y figuras de objetos. Las posibilidades de los ordenadores personales como el PC abrieron las puertas a nuevos conceptos en juegos de puzzles. Entonces surgió un título ingenioso y único: *Lemmings* (1992). El jugador debe controlar a unos hombrecitos para llevarlos sanos y salvos a la salida del nivel por medio de ciertas habilidades. Poseía unos gráficos y animaciones fantásticos para la época. Rompió con el esquema de juegos de habilidad por el del sistema ensayo - error.

Existen infinidad de títulos para este género. En 2D o 3D, con mejores o peores gráficos, todos siguen la premisa de mantener al jugador o jugadores pegados al monitor resolviendo puzzles y desarrollando la agilidad mental sin ningún tipo de violencia.



En el próximo número...

... hablaremos de los juegos de rol.

Cuestionario Videojuegos



Preguntas

1. ¿Cómo podemos modificar el tamaño de la vista de una cámara en Blitz3D?

2. ¿Cómo podemos saber con el puntero del ratón la posición de un objeto 3D en Blitz3D?

3. ¿Cómo podemos aislar una colisión para saber con qué elemento de un mismo tipo hemos colisionado?

4. ¿Cómo podemos colocar un sprite en un polígono con la misma orientación que éste?

5. En Milkshape3D, ¿qué operación podemos utilizar para modelar la estructura de una roca?

6. Antes de poder pintar en Deep Paint 3D, ¿qué es lo que debe tener creado el modelo?

7. Si queremos grabar una pista MIDI en Anvil Studio con un teclado externo, ¿qué es lo que debemos preparar primero para que sea posible?

8. ¿Cómo podemos cuantizar una pista en Anvil Studio?

9. Define y lee una base de datos que contenga las letras del alfabeto en Blitz3D.

10. ¿Cómo podemos sustituir una letra de una frase por una imagen en Blitz3D?

Respuestas al cuestionario 12

▷ **1.** "LoadAnimMesh" se utiliza para cargar un modelo con o sin animación. En caso de no contener animación, cargaría todas las partes del modelo si éste no estuviera agrupado. En caso contrario, obtendríamos la secuencia de animaciones completa.

▷ **2.**
```
Modelo_hombre = LoadAnimMesh ("hombre_andando.3ds")
Animate Modelo_hombre, 1
```

▷ **3.**
```
;Definimos la estructura de datos
Type tipo_disparo
    Field Sprite
    Field x,y,z
End Type
; llamamos a la función para crear un disparo desde la nave
Crear_Disparo.tipo_disparo(pivote_nave,sprite_disparo)
  ;Funcion para crear el disparo
Function Crear_Disparo.Tipo_disparo(pivote_nave,entidad_disparo)
    Disparo.tipo_disparo = New tipo_disparo
    Disparo\Sprite= CopyEntity (entidad_disparo, pivote_nave)
    Disparo\x= EntityX(pivote_nave)
    Disparo\z= EntityZ(pivote_nave)
    Disparo\y= EntityY(pivote_nave)
End Function
```

▷ **4.** Llamando a la función de actualización para cada disparo con un bucle:
```
For Disparo.tipo_disparo= Each tipo_disparo
    Actualizar_disparo(disparo)
Next
```

▷ **5.** Activando la casilla "IK Chain Terminator" en la ventana de opciones *Tool Option* en *Joint*.

▷ **6.** Modificando el "framerate" manualmente. Para ello, elegimos la opción *Custom* en *Set framerate* del menú *Animation*.

▷ **7.** En Anvil Studio se pueden crear pistas MIDI (instrumentos), de ritmos y pistas de audio.

▷ **8.** Los eventos MIDI en Anvil Studio se editan en la sección *Compose*.

▷ **9.** Utilizando planos, por medio de una primitiva cúbica o utilizando mapeado cúbico.

▷ **10.** En primer lugar debemos preparar las texturas en un programa de dibujo. Seguidamente, modelamos un cubo lado a lado. Cada lado lo formamos uniendo vértices y triángulos en una superficie y que posteriormente texturizamos con un *brush*.

Contenido CD-ROM 13

▶ AUDIO

■ AnalogX Scratch
Pequeño programa gratuito para realizar scratches y luego exportarlos a un fichero .wav.

■ Audacity
Completo editor de audio con el que grabar, mezclar y editar pistas de audio con calidad CD.

■ Groovebox
Maravilloso programita hecho en Flash con el que podrás realizar un montón de mezclas en vivo.



■ JV Percussion Generator Light 4
Genera percusiones a partir de simples palabras. Muy divertido.

■ MbooM
Secuenciador que además permite el tratamiento de melodías.

▶ DISEÑO 2D

■ Advanced JPEG Compressor
Herramienta que nos asiste en la tarea de convertir y comprimir imágenes en formato JPG, para el uso en la web.



■ DiagramStudio 2.2
Con este potente programa podrás crear diagramas para poner en orden tus ideas y proyectos.

■ FireGraphic XP
Solución completa para visualizar, ordenar e imprimir imágenes con rapidez.

■ Photolightning
Sencillo programa para optimizar y realizar cambios rápidos en nuestros archivos gráficos.

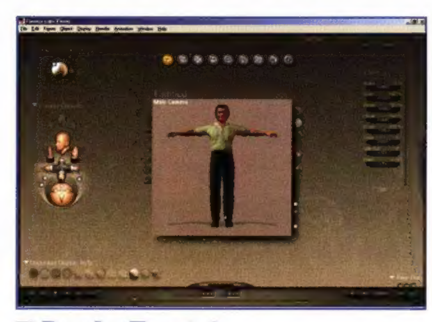
■ PhotoMeister 1.22
Con esta utilidad podrás crear completos álbumes con todas tus fotos e ilustraciones.

■ Tess 1.41
Usando esta herramienta lograrás dibujar formas geométricas en pocos minutos.

▶ DISEÑO 3D

■ D Sculptor 2.0
Editor de objetos 3D muy completo y que produce resultados de una gran profesionalidad.

■ Poser 4
Uno de los programas de diseño 3D más utilizados y potentes con el que podrás modelar espectaculares personajes.

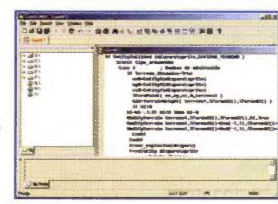


■ RayArtTan 1.0
Render para animaciones en tres dimensiones muy fácil de usar.

■ SolidWorks Property PropaGator
Maneja eficientemente tus archivos de SolidWorks.

▶ PROGRAMACIÓN

■ SuperEditor 2.01
Simplifica tus tareas de programación con este potente editor con el que verás el código mucho más claro.



■ Defect Agent 3.0
Con esta práctica utilidad podrás ir tomando nota de todos los fallos en el desarollo.

■ EasyLicenser License Manager
Protege el software que crees con esta herramienta de control de licencias.

■ Emeditor
Pequeño editor con soporte para múltiples lenguajes de programación.

■ InstallConstruct 5.3
Crea instaladores, agentes de ayuda y desinstaladores para tus programas.

■ Video2000
Evalúa la calidad de tus imágenes y del juego para ver si éste se visualiza correctemente.

▶ JUEGOS

■ Block Out 1.58
Famoso juego que no es sino un peculiar Tetris en tres dimensiones.

■ Bust-A-Move 4
Simpático puzzle en el que debes explotar todas las burbujas a tiempo.

■ Populous The Beginning
Demo del estupendo juego en el que adquirías el poder de ser un dios.

■ Lemmings 3D
Versión en tres dimensiones de los Lemmings, que tantos adeptos sigue teniendo.

■ Tetristation 1.0
Juega al clásico Tetris en esta nueva versión, que sin duda te enganchará.



■ Zone of Fighters
Como todas las semanas, nuestro juego.

▶ VÍDEO

■ Joiner 1.1
Aplicación para unir y sincronizar vídeo y audio muy fácilmente.



■ M2 Edit Pro 5.0
Editor avanzado de MPEG-2 con el que podrás realizar auténticas virguerías.

■ WinMPG Video Convert 1.50
Convierte tus archivos de vídeo en distintos formatos usando este programa.

▶ EXTRAS

En este apartado encontrarás todos los ejemplos de los que hablamos en el coleccionable, para que no pierdas detalle.